# Refactoring GrPPI

Brown, Christopher; Janjic, Vladimir; Barwell, Adam D.; Garcia, J. Daniel; MacKenzie, Kenneth

*Published in:*
International Journal of Parallel Programming

*DOI:*
10.1007/s10766-020-00667-x

*Publication date:*
2020

International Journal of Parallel Programming
https://doi.org/10.1007/s10766-020-00667-x

# Refactoring GrPPI: Generic Refactoring for Generic Parallelism in C++

**Christopher Brown[1] · Vladimir Janjic[2] · Adam D. Barwell[1] · J. Daniel Garcia[3] · Kenneth MacKenzie[4]**

Received: 16 October 2019 / Accepted: 13 June 2020


**Abstract**
The Generic Reusable Parallel Pattern Interface (GrPPI) is a very useful abstraction over different parallel pattern libraries, allowing the programmer to write generic patterned parallel code that can easily be compiled to different backends such as FastFlow, OpenMP, Intel TBB and C++ threads. However, rewriting legacy code to use GrPPI still involves code transformations that can be highly non-trivial, especially for programmers who are not experts in parallelism. This paper describes *software refactorings* to semi-automatically introduce instances of GrPPI patterns into sequential C++ code, as well as *safety checking* static analysis mechanisms which verify that introducing patterns into the code does not introduce concurrency-related bugs such as race conditions. We demonstrate the refactorings and safety-checking mechanisms on four simple benchmark applications, showing that we are able to obtain, with little effort, GrPPI-based parallel versions that accomplish good speedups (comparable to those of manually-produced parallel versions) using different pattern backends.

**Keywords** Refactoring · Parallelism · Parallel patterns · TBB · C++ · GrPPI · C++ threads

## 1 Introduction

The scale of parallelism in modern hardware systems is increasing at a very fast rate, with 72-core systems available off-the-shelf even in the embedded market.[1] At the same time, such systems are becoming increasingly heterogeneous, integrating GPUs, FPGAs, DSLs and other specialised processors within the same chip. The large scale of parallelism and heterogeneity of systems make programming modern parallel hardware very difficult, often requiring a combination of different (and usu-

[1] http://www.mellanox.com/page/products_dyn?product_family=238&mtag=tile_gx72.

✉ Christopher Brown
cmb21@st-andrews.ac.uk

Extended author information available on the last page of the article

Published online: 10 July 2020

ally complex) programming models (e.g. POSIX threads for the central multicore processor and OpenCL for GPUs), coupled with careful manual tuning, to achieve good performance. Parallel patterns [4] have been recognised as an excellent compromise between the ease of programming and the ability to generate efficient code for large-scale heterogeneous parallel architectures. They have been endorsed by several major IT companies, such as Intel [42] and Microsoft [14], giving rise to a multitude of parallel pattern libraries, most of which are incompatible with one another and each of which usually has specific advantages (and disadvantages) over the others. GrPPI[2] [16] represents one of the first attempts at a uniform interface to parallel patterns, based on C++ template programming, that allows the generation of target code for different pattern libraries. Listings 1 and 2 show code for a *farm* pattern (a typical embarrassingly parallel pattern) that targets both OpenMP and Intel TBB pattern libraries. We can observe that the only difference is in the declaration of the `par` variable, in Line 1, where the back-end for implementation is specified. While GrPPI makes the patterned code easier to write [16], and also minimizes the cost of switching between different implementation pattern libraries, transforming sequential code to use GrPPI is still a very non-trivial task. The programmer needs to first ensure that it is *safe* (in terms of unexpected side effects and race conditions) to transform the sequential code into its equivalent parallel implementation, and, secondly, to transform loops of the sequential code into calls to the appropriate patterns. An equivalent version of the code in Listing 2 is given in Listing 12.

This paper describes *refactorings* to introduce GrPPI patterns into sequential code. These refactorings, implemented in the ParaFormance toolset for developing and maintaining parallel programs, provide a semi-automatic way of transforming sequential C++ code into its parallel patterned counterpart. The programmer is only required to insert simple annotations in the code to denote the parts that are, possibly, amenable to patterned parallelisation. We also describe *safety checking* mechanisms that ensure the parts of the code annotated by the programmer are, indeed, safe to be transformed into patterns. Safety checking is based on static analyses of the loops in the application to ensure that refactoring the code does not introduce any undesired behaviour to the execution once parallelised.

[2] https://github.com/arcosuc3m/grppi.

**Listing 1** GrPPI Example targeting OpenMP

```
parallel_execution_omp par{};

pipeline(par,
  [] -> optional<image> {
    image im;
    if (read(im)) return im;
    else return {};
  },
  farm(4,
    [](const image & im) {
      return process_image(im);
    }
  ),
  [](const image & im) {
    write_image(im,file);
  });
```

**Listing 2** GrPPI Example targeting TBB

```
parallel_execution_tbb par{};

pipeline(par,
  [] -> optional<image> {
    image im;
    if (read(im)) return im;
    else return {};
  },
  farm(4,
    [](const image & im) {
      return process_image(im);
    }
  ),
  [](const image & im) {
    write_image(im,file);
  });
```

The specific research contributions of this paper are:

1. Novel refactorings to introduce *farm* and *pipeline* parallelism, based on the GrPPI interface, into sequential C++ code, implemented as part of the *ParaFormance* refactoring tool-set;
2. A study into the process of refactoring sequential C++ applications into their *safe* parallel equivalents, on a range of different real-world examples, for different parallel implementations, using a fully-automated tool-supported refactoring framework;
3. Demonstrations of the effectiveness of the performance of refactored examples, showing speedups of up to 23.93 on a 28-core machine over the sequential code;
4. The transformations of four applications into their GrPPI equivalents, together with a discussion of the transformation methods.

## 2 Patterns, GrPPI and Refactoring

*Parallel patterns* are a high-level abstraction for representing classes of computations that are similar in terms of their parallel structure, but different in terms of problem-specific operations. A typical example of a parallel pattern is a *parallel map*, where the same operation is applied to disjoint subsets of the input in parallel. Regardless of whether the actual operation is, for example, multiplying a matrix by a vector or processing a pixel of an image, the parallel structure of the computation will be the same. Parallel patterns are typically implemented as library functions, which handle creation, synchronisation and communication between parallel threads, while the problem-specific (and often sequential) computations are provided as the pattern parameters. In this paper, we restrict ourselves to two classical parallel patterns, which can be further generalised to include a broader set of parallel patterns.

– The pipeline pattern models a parallel pipeline. Here, a sequence of functions, $f_1, f_2, \ldots, f_m$ are applied, to a stream of independent inputs, $x_1, \ldots, x_n$. The output of $f_i$ becomes the input to $f_{i+1}$, so that the parallelism arises from executing $f_{i+1}(f_i(\ldots f_1(x_k)\ldots))$ in parallel with $f_i(f_{i-1}(\ldots f_1(x_{k+1})\ldots))$. In this case the

parallelism arises from executing different stages $f_i$ in parallel while items progress through the pipeline.

– The farm pattern models a task parallel computation for a stage $f_i$ in a pipeline that can be applied to a stream of independent inputs, $x_1, \ldots, x_n$. For each item $x_j$ in the input stream the farm delivers to the output stream the value $f_i(x_j)$. Multiple applications of the operation to different input stream elements may be processed in parallel.

Details and semantics of these and other patterns are described in [16]. Note that the technique described in this paper can be further generalised to include the full set of commonly-used parallel patterns.

*Refactoring* is the process of changing the structure of a program while preserving its functional semantics in order, for example, to increase code quality, programming productivity and code reuse. The term refactoring was first introduced by Opdyke in his PhD thesis in 1992 [38], and the concept goes at least as far back as the fold/unfold system proposed by Burstall and Darlington in 1977 [13]. In our case, refactorings are source-to-source transformations of the code that are performed *semi-automatically*, under the programmer's guidance, and possibly with their input. ParaFormance [3] is a refactoring tool-suite developed at the University of St Andrews that refactors C and C++ programs into parallel versions. It targets a number of different back-ends, including FastFlow, OpenMP, GrPPI and Intel Threading Building Blocks (TBB).

GrPPI [16] is a parallel pattern interface that uses C++ template metaprogramming to provide implementations of a number of parallel patterns. The patterns in GrPPI are generic over a number of different parallelism models, currently including support for ISO C++ Threads, OpenMP, Intel TBB, and FastFlow, as well as sequential execution. The ability of decoupling patterns from the concrete execution model is a key idea in GrPPI. With GrPPI, the application source code can be the same independently of the concrete execution policy. This approach reduces the conceptual load for programmers as they can focus on how computations are composed without paying attention to details that are specific to a programming model [16]. Moreover, this increases portability as moving from one parallel framework to a different one has no significant impact on source code. Additionally, some models might not be available (or allowed by coding standards or certification policies) in given platforms which is solved in GrPPI by selecting a different one. The ISO C++ Threads back-end is provided as a fallback and is always guaranteed to be present in any ISO C++ compliant platform. In [23] additional evidence on the negligible overhead of GrPPI over manual implementations is provided. GrPPI offers a set of patterns that can be classified in three groups: data patterns, task patterns and stream patterns. In this paper we focus on stream patterns. *Data patterns* perform a transformation on one or more data sets and give as a result a new data set (map, stencil) or a value (reduce or map/reduce). In all those patterns the input dimensionality is unbounded, meaning that the transformation can be applied to any number of data sets. This is different to the C++ standard library, which takes one or two data sets as an input, but similar to the approach of SkePU 2 [21]. The only *task pattern* included in GrPPI is *divide-and-conquer* which allows the expression of computations where a given problem is split (possibly recursively) into

[3] http://www.paraformance.com.

smaller sub-problems which are then solved and combined. GrPPI is able to apply parallelism during the three stages of the pattern. The basic GrPPI model for stream parallelism is a *pipeline* which processes a stream of data items: items are produced by a *generator*, processed by some intermediate components, and then disposed of by a *sink*. The individual stages can all be executed in parallel. A pipeline stage can be any callable entity. However, the most common case is a C++ lambda expression. A simple example is given below that reads a number of integers from a file, squares each one, and writes the results to standard output. The first lambda reads a value from the input file and returns it. If the read operation returns `false` (meaning *end-of-file*) the empty optional is returned. The second stage is a farm replicating its lambda in four independent tasks. Each of them receives a number and returns its square. Finally, the third lambda receives a number and prints it to standard output. Note, that all the communication and synchronization between stages is managed internally by GrPPI.

```
parallel_execution_native par{};

pipeline(par,
   [&]() -> optional<int> {
       int i;
       if (file >> i) return i;
       else return {};
   },
   farm(4,
      [](int k) { return k*k; }
   ),
   [](int n) {
      cout << n << "\n";
   }
);
```

The `parallel_execution_native` definition introduces a GrPPI object that describes the execution model to be used by the parallel pipeline. In this case, the native object is used to indicate that the native execution model (C++ threads) should be used; in order to use, e.g., TBB instead, one would replace this with `parallel_execution_tbb`. Further details on the interfaces and semantics of patterns available in GrPPI can be found in [16].

## 3 Refactoring for Introducing a GrPPI Pipeline

We define a refactoring that converts a C++ `for` loop into a GrPPI pipeline pattern, containing one or more stages that are executed concurrently to process a sequence of data items indexed by the loops. We currently support both sequential stages that process a single data item at a time, and farm stages that process multiple items in parallel. For a singular farm pattern, our current refactoring approach would transform the code into a pipeline with a single stage that is farmed.

### 3.1 Refactoring Strategy

To refactor a `for` loop, the loop must be in the form,

```
for (T x=e1; e2; e3) {...}
```

for some variable, `x`, and type `T`, where `T` may be omitted. As is standard, `e1` is an initial value for `x`, `e2` is some bounding condition, and `e3` is an expression that updates the value of `x`. When `e2` is empty, the pipeline will run forever. The refactoring requires that `e3` must not be empty, and that the loop must declare or initialise a single variable in the initialiser; this variable represents data items passing through the pipeline. These patterns enable the refactoring of common types of loop; e.g.

```
1 for (int i=0; i<N; ++i) { ... }
```

and

```
1 for (auto i = v.begin(); i != v.end; ++i) { ... }
```

The refactoring requires that the body of the `for` loop is a compound statement enclosed in braces, i.e. `{ ... }`, and is dependent upon the existence of pragmas in the loop body that indicate the pipeline stages. These pragmas may be introduced manually by the programmer, or automatically via some tool, which we intend to investigate as part of future work. We note that these pragmas are not part of GrPPI, and do not affect the functional behaviour of the program in any way, but are instead introduced here as an aid to the refactoring. These pragmas include:

- `#pragma grppi seq stage`
- `#pragma grppi farm stage`*n*

A GrPPI farm stage pragma must specify a number of threads to execute the farm, specified by *n*, where *n* is a literal integer or variable name bound to a literal integer. It is possible to create a pipeline that comprises a single farm stage: this is equivalent to running multiple (and possibly all) iterations of the loop concurrently. Once invoked, the refactoring requires the programmer to provide: a name for the pattern to be inserted; the model of parallelism, e.g. C++ threads or TBB; and any additional headers for other modes. The refactoring procedure creates a GrPPI `pipeline` object and inserts a source object that returns consecutive values for `x` in the form of `optional` items, with an empty value when there are no data items left. The pipeline stages in the loop body are converted into a sequence of lambda expressions following the source. For example, the loop,

```
1 for (int i = 0; i < NUM_ELEM; i++) {
2     #pragma grppi seq stage
3     xs[i] = s1(xs[i]);
4     #pragma grppi farm stage 4
5     xs[i] = s2(xs[i]);
6 }
```

which iterates over the array `xs`, applying the composition of `s1` and `s2`, is transformed into

```
parallel_execution_native pipe { };

pipeline(pipe,
    [i=0,max=NUM_ELEM]() mutable -> std::optional<int> { // source
        if (i < max) return i++;
        else return {};
    },
    [&](int i) { // stage 1
        xs[i] = s1(xs[i]);
        return i;
    },
    farm(4, [&](int i) { // stage 2
        xs[i] = s2(xs[i]);
        return;
    }));
```

If a variable is declared inside a loop stage but required in a later stage, it will be returned from the corresponding lambda and passed as an argument to the following lambda: if there is more than one such variable then they will be returned packed into a tuple which will then be unpacked into variables in the next stage. Variables that are declared outside the loop are captured in the lambda expressions *by reference*, which means that the lambdas can modify them. This may lead to race conditions. They can also be captured *by value* when no modification is needed. This option improves thread safety.

### 3.2 Refactoring Observations

This paper presents a full framework for tool-supported parallel programming in C++. GrPPI provides a high-level unified interface to the underlying skeleton implementation, providing a *palette* of parallelisations to execute. Refactoring provides the user with the *choice* and *guidance* to ensure a correct and safe implementation of the skeletal choice. Although GrPPI provides a high-level easily accessible interface to skeletal programming in C++, refactoring is still a necessary step and provides a number of unique benefits over manual parallelisation. Refactoring helps the user make decisions about their program and the appropriate skeletal configuration to choose for their application. Our refactoring support, together with its safety checking, *avoids common pitfalls* in parallel programming, such as *introducing deadlocks* and *race-conditions*, which are notoriously difficult and subtle errors to find and repair. Refactoring follows very precise (and well-understood) transformation rules that are based on well known semantics-preserving program rewriting techniques, ensuring that *only correct programs can be derived*, saving the programmer time and effort in fixing bugs. The idea of annotating the source program with annotations (or pragmas) is different from previous refactoring approaches of introducing farms and pipelines [29]. Identifying the stages and/or components of the skeletons using pragmas also allows for further tool-support to *discover* candidates for parallelism [10]. Such static analysis techniques (such as those in [17]) can identify the components of the skeletons and insert annotations into the source code that the refactorings can then use.

## 4 Safety Checking

The refactorings presented in this paper should preserve the correctness of the functional semantics of the C++ program. This means that when given the same input value(s), the program should produce the same output value(s) before and after a refactoring, up to a given ordering. This is ensured by *safety checking*, a fundamental feature in the ParaFormance tool. It gives confidence that the code being refactored is safe for parallelisation, meaning that it is free from dependencies, side effects (such as writing to a global state) and that it does not contain code that may interfere with the parallel logic. In the ParaFormance tool, we centred the safety checking around *Array Race Detection Analysis*, a common technique that is based on Pugh's Omega Test Library [39] and standard compiler techniques, such as data dependency analysis and those described in [2] and [37]. We give some details of our implementation in this section. Some terminology:

- An *index* means a loop control variable like `i` in `for (int i=0; ...)`.
- A *subscript* means an expression like `e` or `4` appearing in `A[e][4]`.
- A *linear expression* is an expression of the form $e_1 * i_1 + \ldots + e_n * i_n + k$ where $e_1, \ldots, e_n$ and $k$ are constants and $i_1, \ldots i_n$ are loop variables.

Our basic problem is to detect when two array accesses `A[e1]` and `A[e2]` access the same element of an array during different iterations of a loop nest, where `e1` and `e2` are expressions that may depend on the loop control variables. For example, in

```
1 for (int i=0; i<10; i++) A[i] += A[i+1];
```

the iterations at `i=4` and `i=5` both access `A[5]`, leading to a possible data race if we execute the iterations in parallel. Similarly, if we have

```
1 for (int i=0; i<10; i++)
2   for (int j=0; j<10; j++)
3     A[i+j] += 1;
```

and we execute the iterations of the outer loop in parallel we will write to (for example) `A[4]` at $(i, j) = (0, 4), (1, 3), (2, 2), (3, 1)$, and $(4, 0)$. Our strategy is as follows. Given a loop, we search for all array accesses within it, saving the subscript expressions. We then compare all pairs of subscripts (apart from the case where both accesses are reads, since simultaneous reads do not lead to a data race). For accesses with non-linear subscripts (e.g. `A[i*i]`, `A[f(i)]`, `A[B[i]]`), it is generally undecidable whether two subscripts are identical, so we give up immediately, reporting a potential race. We allow multidimensional accesses like `A[i+j][j+2*k][k]`, where we look for simultaneous solutions for corresponding pairs of subscripts. If the subscripts are linear expressions with constant coefficients (`A[2*i+7*j-k+1]`, for example), then we use the Omega library [39] to check for satisfiability. If a race can occur, we use the Omega library to give example values of index variables causing a race. If the loop bounds are also simple linear expressions, then we include them in the data for the test and we get a precise result: the test will always tell us whether or not a race condition occurs. If the bounds are not simple linear expressions (for example, `v.size()`), then we leave the corresponding variable unbounded and the Omega test's response will be conservative. For example, if we have

```
1 for (i=0; i<=i*i; i++) A[i] += A[2];
```

then the loop will only execute with `i=0` and `i=1`, so there cannot be a race with `A[2]`. However, our test assumes that `i` can take any positive value, and so will report a possible data race at `i=2` (even though the user may be able to see that we never get to `i=2`). If the loops are linear expressions with non-literal coefficients, like `A[M*i+N]` where `M` and `N` are variables, then we fall back on a symbolic test. We allow loops of the form

```
1 for (T i=e1; i op e2; incr) ...
```

where `op` is `<`, `<=` and `incr` is `i++`, `++i`, `i+=n`, or `i=i+n`. The index variable can be of any integral type `T`, and one can also use index variables which are declared outside the loop header. The lower and upper bounds `e1` and `e2` can be arbitrary expressions, but we get better results if they are fairly simple (for example, known integers). The Omega test will correctly handle literal strides greater than `1`; for example, in

```
1 for (i=0; i<N; i+=10) for (j=0; j<10; j++) A[i+j]++;
```

there will not be a race condition, but there will be one if `i+=9` is put instead. Other safety checks that we implement as part of this framework will be described in future work.

## 5 Evaluation

In this section, we present an evaluation of the refactorings to introduce GrPPI patterns into sequential code. We consider four benchmark applications: Mandelbrot, Matrix Multiplication, Ant Colony Optimisation and Image Convolution. As described in Table 1, the benchmark applications belong to different domains and also contain different compositions of parallel patterns, hence the refactorings applied for their parallelisation are different. In addition, refactoring some of them exposes safety problems that are caught by the safety analysis, whereas the refactoring process for others is more straightforward. For each, we start with a given sequential version of a benchmark. We then use the refactorings described in Sect. 3 to introduce GrPPI patterns, using interfaces to C++ threads and TBB (*GrPPI Native* and *GrPPI TBB*, respectively, in the graphs below). This produces the refactored parallel versions. To measure the performance of these versions, we compare them with the *manually-produced* parallel versions of the baseline applications (*Par Manual* in the graphs below). These versions have been written by hand using C++ threads and are highly optimised. Our goal is to verify how the execution time of the refactored parallel versions compare with good hand-produced parallel code. All of our execution experiments are conducted on a server with a 28-core Intel Xeon E5-2690 CPU running at 2.6 GHz, with 256 GB of RAM, and the Scientific Linux 6.2 operating system.

**Table 1** Benchmarks used for evaluation

| Benchmark | Application area | Type of parallelism |
|---|---|---|
| Mandelbrot | Mathematical visualisation | Farm |
| Matrix multiplication | Computation mathematics | Farm |
| Image convolution | Image processing | Pipeline with farms |
| Ant colony optimisation | Evolutionary computing | Pipeline with farm |

### 5.1 Mandelbrot

Mandelbrot is a simple benchmark that calculates a Mandelbrot set for a set of points in complex plane and visualises it. A point $\mathcal{C}$ from the complex plane is in the Mandelbrot set if the orbit $z_n$ of the point, obtained using a recurrence relation $z_{n+1} = z_n^2 + \mathcal{C}$, does not tend to infinity. The set can be visualised by colouring the points in the complex plane based on the number of steps of the recursive relation required to reach the maximum radius. The relevant part of the annotated original sequential version is given in Listing 3.

**Listing 3** Sequential Mandelbrot

```
1  template <typename F>
2  void get_number_iterations(window<int> & scr, window<double> & fract, int
       iter_max,
3                             std::vector<int> & colors, F func) {
4        int k = 0;
5        int N = scr.width();
6        for(int i = scr.y_min(); i < scr.y_max(); ++i) {
7  #pragma grppi farm stage 24
8              for(int j = scr.x_min(); j < scr.x_max(); ++j) {
9                    complex<double> c(j, i);
10                   c = scale(scr, fract, c);
11                   colors[k++] = escape(c, iter_max, func);
12             }
13       }
14 }
```

Note that the pragma at Line 7 denotes the loop on the Lines 8–13 as a candidate for parallelisation using the farm pattern. However, if we try to refactor the aforementioned loop by replacing it with an instance of the farm pattern, the safety checking analysis recognises that there is a global variable `k` that is incremented in each iteration of the loop and that, therefore, straightforward parallelisation of this loop using the farm pattern would introduce race conditions. The ParaFormance tool also suggests the rewriting of the code into the equivalent version which avoids the above problem by induction variable substitution; i.e. replacing the global variable `k` inside the loop with a local one that is calculated based on the loop index. The listing in Figure 4 shows this version.

**Listing 4** Improved Sequential Mandelbrot

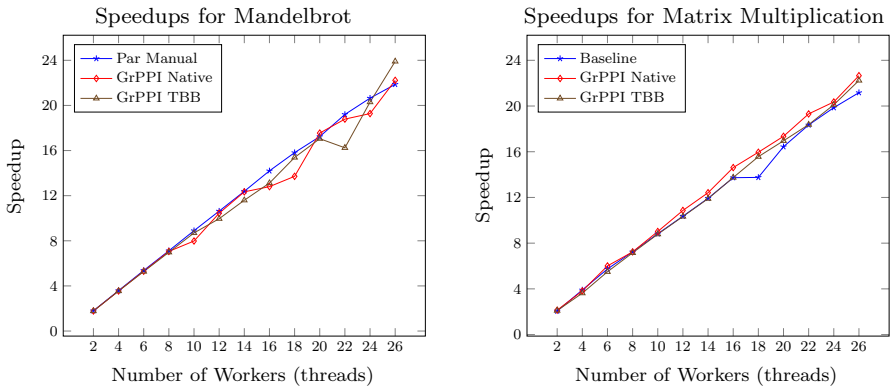
```
1  template <typename F>
2  void get_number_iterations(window<int> & scr, window<double> & fract, int
       iter_max,
3                             std::vector<int> & colors, F func) {
4      int N = scr.width();
5      for(int i = scr.y_min(); i < scr.y_max(); ++i) {
6  #pragma grppi farm stage 24
7          for(int j = scr.x_min(); j < scr.x_max(); ++j) {
8              complex<double> c(j, i);
9              c = scale(scr, fract, c);
10             int Ni = N*i+j;
11             colors[Ni] = escape(c, iter_max, func);
12         }
13     }
14 }
```

Note that the global variable `k` has been replaced with the local variable `Ni`. The loop on lines 7–12 is now safe for refactoring. Invoking the 'Introduce GrPPI Pipeline' refactoring from the ParaFormance tool then gives the following GrPPI native version of the code

**Listing 5** GrPPI Native Mandelbrot

```
1  template <typename F>
2  void get_number_iterations(window<int> & scr, window<double> & fract, int
       iter_max,
3                             std::vector<int> & colors, F func) {
4      parallel_execution_native pipe {};
5
6      pipeline(pipe,
7          [i=scr.y_min(),max=scr.y_max()]() mutable -> optional<int> {
8              if (i < max) return i++;
9              else return {};
10         },
11         farm(24, [&,N=scr.width()](int i) {
12             for (int j = scr.x_min(); j < scr.x_max(); ++j) {
13                 complex<double> c(j, i);
14                 c = scale(scr, fract, c);
15                 int Ni = N*i+j;
16                 colors[Ni] = escape(c, iter_max, func);
17             }
18         }
19     ));
20 }
```

As explained in Sect. 3, the GrPPI TBB version of the code can easily be obtained by replacing `parallel_execution_native` with `parallel_execution_tbb` on line 4 of Listing 5. Finally, Listing 6 shows the manually produced parallel version of the code, using the TBB library.




**(a)** Speedup results for Mandelbrot

**(b)** Speedup results for Matrix Multiplication

**Fig. 1** Speedup results for Mandelbrot and Matrix Multiplication

**Listing 6** Manual Parallel Mandelbrot

```
template <typename F>
void get_number_iterations(window<int> & scr, window<double> & fract, int
     iter_max,
                           std::vector<int> & colors, F func) {
       int k = 0;
       int N = scr.width();
       parallel_for(blocked_range<int>(scr.y_min(), scr.y_max()),
            [&](const blocked_range<int> &range) {
                for (int i = range.begin();i != range.end();++i) {
                    for(int j = scr.x_min(); j < scr.x_max(); ++j) {
                        complex<double> c(j, i);
                        c = scale(scr, fract, c);
                        int Ni = N*i+j;
                        colors[Ni] = escape(c, iter_max, func);
                    }
                }
            });
  }
```

Figure 1a shows the speedups obtained for the GrPPI Native, GrPPI TBB and Manual Parallelisation versions of the code, with respect to the number of workers used in the *farm* pattern of the GrPPI and manual TBB versions. We can observe that all the versions give very good and comparable results, so in this case the refactored version of the code produced semi-automatically is as good in terms of performance as is the manually produced parallel version.

### 5.2 Matrix Multiplication

Matrix multiplication is one of the most commonly used simple parallel benchmarks that demonstrates the use of the *map* or *farm* pattern. Listing 7 shows the sequential version of the benchmark that multiplies matrix $a$ with matrix $b$ and stores the result in $res$.

**Listing 7** Sequential Matrix Multiplication

```
1  double multiply_row_by_column(const matrix & mat1, const matrix & mat2,
2                                int row, int col) {
3    double sum = 0;
4    for (int k = 0; k < mat2.rows(); k++)
5      sum += mat1(row, k) * mat2(k, col);
6    return sum;
7  }
8
9  void multiply_row_by_matrix(const matrix & mat1, const matrix & mat2, matrix &
       res,
10                             int row) {
11   for (int col = 0; col < res.rows(); col++)
12     res(row, col) = multiply_row_by_column(mat1, mat2, row, col);
13 }
14
15 matrix matrix_multiply(const matrix & a, const matrix & b) {
16   matrix res{a.rows()};
17   for (int i = 0; i < a.rows(); i++)
18     multiply_row_by_matrix(a, b, res, i);
19   return res;
20 }
```

We can parallelise this version by assigning a separate task/thread to each call to the `multiply_row_by_column` function, as these calls are completely independent of each other. This would, however, create $n \times n$ tasks/threads for multiplying two $n \times n$ matrices, making the parallelisation too fine-grained (and, indeed, infeasible if the C++ threads are used for parallelisation). Therefore, both in our hand-tuned baseline parallelisation using C++ threads and in the GrPPI version, we parallelise only the loop in the `matrix_multiply` function (Line 16 in Listing 7), assigning separate tasks for each call to the `multiply_row_by_matrix` function. In the baseline version, shown in Listing 8, we further use *chunking* to increase granularity, grouping multiple calls to the function into a single thread, so that we have exactly as many threads as there are, for example, cores on a multicore machine where the application is executed, and each of them executes a series of calls to the `multiply_row_by_matrix` function.

**Listing 8** Baseline (C++ threads) Parallel Matrix Multiplication

```
1  // This is a chunked multiply_row_by_matrix function
2  void multiply_range(const matrix & a, const matrix & b, matrix & res,
3                      int row_start, int row_end) {
4    for (int i = row_start; i < row_end; i++)
5      multiply_row_by_matrix(a, b, res, i);
6  }
7
8  void matrix_multiply(const matrix & a, const matrix & b, matrix & res, int
       nthreads) {
9    using namespace std;
10   int chunk_size = a.rows() / nthreads;
11   vector<thread> tasks;
12
13   for (int i = 0; i < nthreads; i++) {
14     int start = i * chunk_size;
15     int end = (i == nthreads - 1) ? a.rows() : (i + 1) * chunk_size;
16     tasks.emplace_back(multiply_range, cref(a), cref(b), ref(res), start, end);
17   }
18   for (auto & t : tasks) { t.join(); }
19 }
```

Listing 9 shows the GrPPI version of the application, obtained automatically from the sequential version using the GrPPI refactorings. The code is for the native (C++ threads) parallelisation based on GrPPI.

**Listing 9** GrPPI Matrix Multiplication

```
1  matrix matrix_multiply(const matrix & a, const matrix & b) {
2    using namespace grppi;
3    matrix res{a.rows()};
4    parallel_execution_native par;
5
6    pipeline(par,
7        [i = 0, max = a.rows()]() mutable -> std::optional<int> {
8          if (i < max) { return i++; }
9          else { return {}; }
10       },
11       farm(4, [&](int i) {
12         multiply_row_by_matrix(a, b, res, i);
13       })
14   );
15   return res;
16 }
```

As explained in Sect. 2, to derive the OpenMP or TBB versions, we would just need to replace the `parallel_execution_native` with `parallel_execution_openmp` or `parallel_execution_tbb`, respectively. This assigns a separate task to each call to the `multiply_row_by_matrix` function. Note that we do not use chunking in this version, as this would require use of the *map* pattern which is present in GrPPI, but this is outside of the scope of this paper. Note also that this example does not introduce any safety problems, as there are no race conditions.

Figure 1 shows the speedups obtained for the Baseline, GrPPI Native and GrPPI TBB versions of the code, with respect to the number of workers used in the *farm* pattern of the GrPPI versions and the number of threads used in the baseline version. We can note that all versions give very good speedups, which are comparable to the GrPPI Native version; the GrPPI TBB being slightly faster than the Baseline version. However, it is worth noting that the GrPPI Native and GrPPI TBB versions use one thread more than the Baseline version, because there is a separate thread assigned to the first stage of pipeline (Lines 6–10 in Listing 9). Therefore, speedups when the same number of threads are used would be approximately the same.

### 5.3 Image Convolution

Image convolution is a technique widely used in image processing applications for blurring, smoothing and edge detection. We consider an instance of the image convolution from video processing applications, where we are given a list of images that are first read from a file and then processed by applying a filter. Applying a filter to an image consists of computing a scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels:

$$out(i, j) = \sum_m \sum_n in(i - n, j - m) \times filt(n, m) \tag{1}$$

An obvious parallelisation of this algorithm is to set up a pipeline where the first stage reads images from a file and the second stage applies the filter to the read images. Each of the stages can be further farmed, so that we can read and process multiple images at the same time. An alternative parallelisation is to set up a farm, where each worker first reads an image from a file and then processes that image. The former parallelisation is better if we need to have a different number of workers in the two farms, i.e. if reading images is notably slower than their processing, whereas the latter one is better if the two operations are of approximately the same computational cost. To demonstrate the refactoring of this example into parallelised GrPPI versions, we start with the original sequential version, below, in Listing 10, refactoring it to a single farm, annotated with a GrPPI pragma indicating a single candidate exists for parallelisation using the farm pattern (Line 2).

**Listing 10** Sequential Image Convolution

```
1 for (int i = 0; i < nr_images ; i++) {
2 #pragma grppi farm stage 24
3    string_p image_name_p ;
4    image_name_p = get_image_name(N[i]);
5    task_t task = read_image_and_mask(image_name_p);
6    out_images[i] = process_image(task);
7 }
```

Running this through the ParaFormance tool, we are able to refactor the code into the GrPPI parallelisation as shown in Listing 11. In this example, the code passes the safety-checking phase of the refactoring process, due to the fact that Image Convolution is a classical parallelisation example, being a relatively straightforward application to parallelise.

**Listing 11** GrPPI TBB Farm Image Convolution

```
1  tbb::task_scheduler_init init(nw);
2  parallel_execution_tbb pipe {};
3
4  pipeline(pipe,
5      [i=0, max=nr_images]() mutable -> optional<int> {
6          if (i < max) return i++;
7          else return {};
8      },
9      farm(nw, [&](int i) {
10         image_name_p = get_image_name(N[i]);
11         task_t task = read_image_and_mask(image_name_p);
12         out_images[i] = process_image(task);
13     }
14 }));
```

Speedup results for the GrPPI farm version of Image Convolution are shown in Fig. 3a, where we show speedups for GrPPI versions utilising both the native backend and the TBB backend. Furthermore, we produce a native TBB version, shown below, in Listing 12. Figure 3a compares all versions of the farmed application with comparable results.

**Listing 12** Native TBB Farm Image Convolution

```
tbb::task_scheduler_init init(NW);

parallel_for(blocked_range<int>(0, nr_images),
    [&](const blocked_range<int> &range) {
    for (int i = range.begin();i != range.end();++i) {
        string_p image_name_p;
        image_name_p = get_image_name(N[i]);
        task_t task = read_image_and_mask(image_name_p);
        out_images[i] = process_image(task);
    }
```

One benefit of refactoring instead of manual parallelisation, is that it offers one easily the choice of different parallelisations. For example, Image Convolution is also an example where, instead of a typical parallelisation using a farm, we can, instead, parallelise with a pipeline, farming different stages to attempt to increase the parallelisation. This can be achieved by returning to the sequential application, by choosing *undo* from the ParaFormance refactoring menu, and then adjusting the GrPPI pragmas, so that the stages of the pipeline are properly outlined, as shown below, in Listing 13.

**Listing 13** Sequential Image Convolution

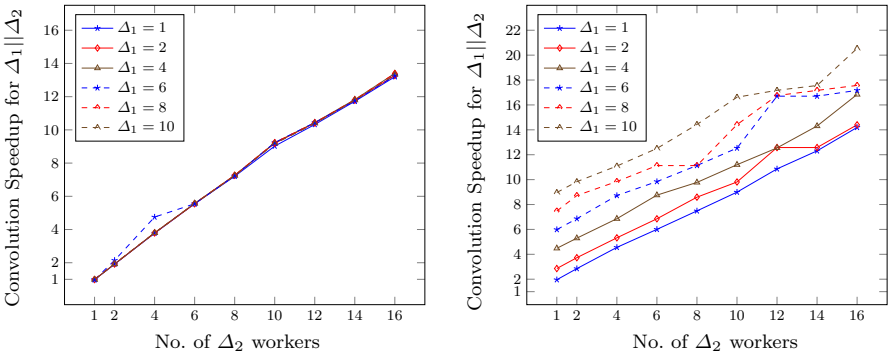
```
1 for (int i = 0; i < nr_images ; i++ {
2     string_p image_name_p ;
3     image_name_p = get_image_name(N[i]);
4 #pragma grppi farm stage farm1
5     task_t task = read_image_and_mask(image_name_p);
6 #pragma grppi farm stage farm2
7     out_images[i] = process_image(task);
8 }
```

Here, we annotate the sequential version with two pragmas: one, at Line 4, indicating a farm stage for the computation `read_image_and_mask`, and a further one at Line 6 for the computation `process_image`. For both of these pragmas, we use defined variables, `farm1` and `farm2`, indicating the number of farm workers for each stage of the pipeline. The result of the refactored code is shown below, in Listing 14.

**Listing 14** GrPPI TBB Pipeline Image Convolution

```
tbb::task_scheduler_init init(farm1+farm2);
parallel_execution_tbb pipe { };

pipeline(pipe,
    [i=0,max=nr_images]() mutable -> optional<int> {
        if (i < max) return i++;
        else return {};
    },
    [&](int i) {
        image_name_p = get_image_name(N[i]);
        return std::make_tuple(i, image_name_p);
    },
    farm(farm1, [&](std::tuple<int,string_p> _args) {
        int i = std::get<0>(_args);
        string_p image_name_p = std::get<1>(_args)
        task_t task = read_image_and_mask(image_name_p);
        return std::make_tuple(i, task);
    }),
    farm(farm2, [&](std::tuple<int, task_t> _args) {
        int i = std::get<0>(_args);
        task_t task = std::get<1>(_args);
```




**(a)** Image Convolution with GrPPI Native pipeline

**(b)** Speedups for Image Convolution with GrPPI TBB Pipeline

**Fig. 2** Speedup results for image convolution

```
        out_images[i] = process_image(task);
    }));
```

Speedup results for the Image Convolution are shown in Figs. 2a (for the GrPPI Native pipeline version), 2b (for the GrPPI TBB pipeline version) and 3a (for the farm version, including the baseline TBB parallelisation). In Figs. 2a, b, each dimension of the graphs shows a varying number of workers for the farm in the first pipeline stage ($\Delta_1$) and the the x-axis of the graph shows the increasing number of workers for the farm in the second stage ($\Delta_2$). Here, we obtain speedups of around 13.98 for 2 $\Delta_1$ workers and 16 $\Delta_2$ workers. We can observe good speedups of up to 13.98 for the GrPPI Native pipeline, 21.23 for the GrPPI TBB pipeline version and 21.43 for the GrPPI Native farm version. We can also observe in Fig. 3a that the GrPPI versions perform approximately the same as the native TBB version, giving almost the same speedups.

### 5.4 Ant Colony Optimisation

Ant Colony Optimisation (ACO) [20] is a metaheuristic used for solving NP-hard combinatorial optimisation problems. In this paper, we apply ACO to the Single Machine Total Weighted Tardiness Problem (SMTWTP) optimisation problem, where we are given $n$ jobs and each job, $i$, is characterised by its processing time, $p_i$, deadline, $d_i$, and weight, $w_i$. The goal is to find the schedule of jobs that minimises the total weighted *tardiness*, defined as

$$\sum w_i \cdot \max\{0, C_i - d_i\}$$

where $C_i$ is the completion time of the job, $i$. The ACO solution to the SMTWTP problem consists of a number of iterations, where in each iteration each ant independently computes a schedule, and is biased by a *pheromone trail* that is stronger along previously successful routes. After all the ants have finished computing solutions in

one iteration, results are gathered, the new best one is picked, the pheromone trail is updated accordingly and the next iteration starts. The relevant part of the original sequential code is given in Listing 15, with the addition of a pragma on Line 3.

**Listing 15** Sequential Ant Colony Optimisation

```
1 for (int j=0; j<num_iter; j++) {
2     for (int i=0; i<num_ants; i++) {
3 #pragma grppi farm stage 24
4       cost[i] = solve (i);
5     }
6     best_t = pick_best(cost, &best_result);
7     update(best_t, best_result);
8 }
```

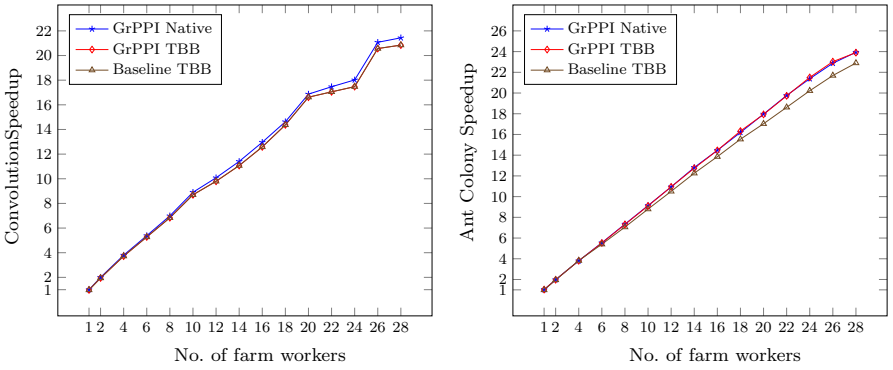
One of the parallelisations of this algorithm is to set up a sequential pipeline, where ants compute solutions in the first stage, and in the second stage, the new running best solution is picked and the pheromone trail is updated. Note that the second stage is inherently sequential, but the first stage can be farmed, assigning a separate task for each ant. This is the parallelisation that we used. The speedup results of which are shown in Fig. 3. We can observe a speedup up to 23.93 for the GrPPI Native version with 28 workers and a comparable speedup of 23.90 with the GrPPI TBB version, also for 28 workers. We compare this to a baseline TBB parallelisation that doesn't use GrPPI and that gives comparable speedups of 22.89. The conclusions here are the same as for the previous examples: the refactored GrPPI versions give good speedups that are approximately the same as manually produced parallel versions. The refactored GrPPI version is shown in Listing 16.

**Listing 16** Native GrPPI Ant Colony Optimisation

```
1  parallel_execution_native pipe {};
2
3  pipeline(pipe,
4      [j=0,max=num_ants]() mutable -> optional<int> {
5          if (j<num_ants) return j++;
6          else {};
7      },
8      farm(24, [&](int j) {
9          cost[j] = solve (j);
10     }));
11
12 best_t = pick_best(cost,&best_result);
13 update(best_t, best_result);
```

## 6 Related Work

Refactoring has roots in Burstall and Darlington's fold/unfold system [13], and has been applied to a wide range of applications as an approach to program transformation [35], with refactoring tools a feature of popular IDEs including, *i.a.*, Eclipse [22] and Visual Studio [36]. Previous work on parallelisation *via* refactoring has primarily focussed on the introduction and manipulation of parallel pattern libraries in C++ [11,29] and Erlang [6,9]. Another approach has been the automated introduction of annotations in the form of C++ attributes [17]. Parallel design patterns, or algorithmic skeletons, were suggested as solution to the difficulties presented by low-level




**(a)** Speedups for Image Convolution GrPPIF arm

**(b)** Speedups for Ant Colony

**Fig. 3** Speedup results for image convolution and ant colony

approaches [4,24]. A range of pattern/skeleton implementations have been developed for a number of programming languages; these include: RPL [29]; Feldspar [5]; FastFlow [1]; Microsoft's Parallel Patterns Library [14]; and Intel's Threading Building Blocks (TBB) library [42]. Since patterns are well-defined, rewrites can be used to automatically explore the space of equivalent patterns, e.g. optimising for performance [26,34] or generating optimised code as part of a DSL [25]. Moreover, since patterns are architecture-agnostic, patterns have been similarly implemented for multiple architectures [28,41]. This introduces a level of specialisation, and the possibility of choice between pattern implementations. Conversely, GrPPI [16] is capable of invoking other libraries, and is thereby able to take advantage of the specialisations that they present without potentially laborious reimplementation. Elsewhere, approaches to automatic parallelisation have traditionally focussed on the transformation of loops. Examples include Lamport's early approaches in Fortran [30], Artigas' approach for Java [2], on DOALL and DOACROSS loops [12,33], the polyhedral model [3,7,8] and more recently on the generation of pipelines [45,46]. Other approaches to automatic parallelism have included a focus on coarsely dividing programs into sections that can be run in parallel [32,43]; less-abstractly on exploiting potential parallelism at the instruction-level [44]; and on exploiting specialised hardware such as GPUs for automatic parallelisation [27,31]. Whilst fully automatic approaches simplify the parallelisation process for the programmer by removing them from the process, such approaches can be very specific in both the parallelism they are able to introduce and the code to which they can be applied. Conversely, programmer-in-the-loop approaches, such as refactoring, allow the programmer to employ their knowledge about both code and parallelism. Similar to our approach, Dig *et al.* [18] use refactoring to introduce parallelism in Java. However, unlike our approach, Dig *et al.* introduce low-level Java concurrency primitives instead of patterns. More recently, Radoi and Dig consider data races in Java for parallelism, a key aspect of safety checking [40]. Other safety checking aspects are covered by work on deadlock detection [15]. PPAT [19], is a parallel pattern identification tool, that uses static analysis and instrumentation to find pipeline and farm

patterns. PPAT is designed as an offline (i.e. manual) refactoring framework with no support for interactive refactoring. Another notable difference with our approach is the integration of safety checks, which are not considered by PPAT. Additionally, while PPAT uses instrumentation to perform dynamic analysis and then update annotations used for refactoring, we avoid the need to instrument and rerun the application by allowing developers to insert annotations in the form of pragmas that will be understood by the refactoring framework.

## 7 Conclusions and Future Work

In this paper, we presented new refactorings for C++ that transform sequential code into fully parallel equivalent implementations using the GrPPI framework. These refactorings are implemented in the ParaFormance tool. Targeting GrPPI allows the programmer to refactor their sequential C++ code into one parallel version that targets many different backends, such as C++ threads, TBB Fastflow, and OpenMP, without having to be a domain expert in parallel programming, or have expertise or knowledge in any of the available parallel libraries. We also presented safety checking mechanisms that ensure the applied refactorings are correct; i.e. that they do not break the semantics of the sequential code. We also demonstrated that we are able to derive good parallel code with the refactorings, achieving speedups similar to the hand-tuned parallel versions. This shows that we are able to produce, with little programming effort, scalable and portable parallel code. In future, we plan to extend our refactorings and safety checking techniques further, to support additional patterns, such as stencil, divide and conquer and reduce. We also plan to evaluate the refactorings on larger use-cases.

**Acknowledgements** This work was supported by the EU Horizon 2020 project, TeamPlay (https://www.teamplay-xh2020.eu), Grant Number 779882, UK EPSRC Discovery, grant number EP/P020631/1, and Madrid Regional Government, CABAHLA-CM (ConvergenciA Big dAta-Hpc: de Los sensores a las Aplicaciones) Grant Number S2018/TCS-4423 .



## References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multicore. In: Programming Multi-core and Many-core Computing Systems (2017)
2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, Burlington (2001)
3. Ancourt, C., Irigoin, F.: Scanning polyhedra with DO loops. In: PPOPP, pp. 39–50, ACM (1991)

4. Asanovic, K., Bodík, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D.A., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.A.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
5. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The Design and implementation of Feldspar—an embedded language for digital signal processing. In: IFL, Lecture Notes in Computer Science, vol 6647, pp 121–136, Springer (2010)
6. Barwell, A.D., Brown, C., Hammond, K., Turek, W., Byrski, A.: Using program shaping and algorithmic skeletons to parallelise an evolutionary multi-agent system in Erlang. Comput. Inform. **35**(4), 792–818 (2016)
7. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: IEEE PACT, IEEE Computer Society, pp. 7–16 (2004)
8. Boulet, P., Feautrier, P.: Scanning Polyhedra without do-Loops. In: IEEE PACT, IEEE Computer Society, pp. 4–11 (1998)
9. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel Erlang programs. Int. J. Parallel Program. **42**(4), 564–582 (2014)
10. Brown, C., Janjic, V., Barwell, A., Thomson, J., Castaneda Lozano, R., Cole, M., Franke, B., Garcia-Sanchez, J., Del Rio Astorga, D., MacKenzie, K.: A hybrid approach to parallel pattern discovery in C++. In: Proceedings of the 28th Euromicro International Conference on Parallel, Distributed and Network-base Processing (2019)
11. Brown, C., Janjic, V., Hammond, K., Schöner, H., Idrees, K., Glass, C.W.: Agricultural reform: more efficient farming using advanced parallel refactoring tools. In: PDP, IEEE Computer Society, pp. 36–43 (2014)
12. Burke, M.G., Cytron, R.: Interprocedural dependence analysis and parallelization (with Retrospective). In: Best of PLDI, ACM, pp. 139–154 (1986)
13. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977). https://doi.org/10.1145/321992.321996
14. Campbell, C., Miller, A.: A parallel programming with Microsoft Visual C++: design patterns for decomposition and coordination on multicore architectures, 1st edn. Microsoft Press, Redmond (2011)
15. Corbett, J.C.: Evaluating deadlock detection methods for concurrent software. IEEE Trans. Softw. Eng. **22**(3), 161–180 (1996)
16. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurr. Comput. Pract. Exp. **29**(24), e4175 (2017)
17. del Rio Astorga, D., Dolz, M.F., Sánchez, L.M., García, J.D., Danelutto, M., Torquati, M.: Finding parallel patterns through static analysis in C++ applications. IJHPCA **32**(6), 779–788 (2018)
18. Dig, D.: A refactoring approach to parallelism. IEEE Softw. **28**(1), 17–22 (2011)
19. Dolz, M.F., del Rio Astorga, D., Fernández, J., García, J.D., Carretero, J.: Towards automatic parallelization of stream processing applications. IEEE Access **6**, 39944–39961 (2018)
20. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press, Cambridge (2004)
21. Ernstsson, A., Li, L., Kessler, C.: SkePU 2: flexible and type-safe skeleton programming for heterogeneous parallel systems. Int. J. Parallel Program. **46**(1), 62–80 (2017)
22. Foundation, E.: Eclipse—an open development platform (2009). http://www.eclipse.org
23. Garcia, J.D., del Rio, D., Aldinucci, M., Tordini, F., Danelutto, M., Mencagli, G., Torquati, M.: Challenging the abstraction penalty in parallel patterns libraries. J. Supercomput. (2019). https://doi.org/10.1007/s11227-019-02826-5
24. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw. Pract. Exper. **40**(12), 1135–1160 (2010)
25. Gorlatch, S.: Domain-specific optimizations of composed parallel components. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. Lecture Notes in Computer Science, vol. 3016. Springer, Berlin (2004)
26. Gorlatch, S., Wedler, C., Lengauer, C.: Optimization rules for programming with collective operations. In: IPPS/SPDP, IEEE Computer Society, pp. 492–499 (1999)
27. Guo, J., Thiyagalingam, J., Scholz, S.: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: DAMP, ACM, pp. 15–24 (2011)
28. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: CGO, ACM, pp. 100–112 (2018)

29. Janjic, V., Brown, C., Mackenzie, K., Hammond, K., Danelutto, M., Aldinucci, M., García, J.D.: RPL: a domain-specific language for designing and implementing parallel C++ applications. In: PDP, IEEE Computer Society, pp. 288–295 (2016)
30. Lamport, L.: The parallel execution of DO loops. Commun. ACM **17**(2), 83–93 (1974)
31. Leung, A., Lhoták, O., Lashari, G.: Automatic parallelization for graphics processing units. In: PPPJ, ACM, pp. 91–100 (2009)
32. Li, H., Thompson, S.J.: Safe Concurrency Introduction through Slicing. In: PEPM, ACM, pp 103–113 (2015)
33. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: POPL, ACM Press, pp. 201–214 (1997)
34. Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z., Akashi, Y.: A fusion-embedded skeleton library. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par. Lecture notes in computer science, vol. 3149. Springer, Berlin (2004)
35. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
36. Microsoft: Visual Studio IDE (2019). https://visualstudio.microsoft.com/vs/
37. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, Burlington (1997)
38. Opdyke, W.F.: Refactoring object-oriented frameworks. In: Ph.D. Thesis, University of Illinois at Urbana-Champaign, Champaign (1992)
39. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: SC, ACM, pp. 4–13 (1991)
40. Radoi, C., Dig, D.: Effective techniques for static race detection in java parallel loops. ACM Trans. Softw. Eng. Methodol. **24**(4), 24:1–24:30 (2015)
41. Reyes, R., Lomüller, V.: SYCL: single-source C++ accelerator programming. In: PARCO, Advances in Parallel Computing, IOS Press, vol 27, pp 673–682 (2015)
42. Robinson, A.: TBB (Intel Threading Building Blocks). In: Encyclopedia of Parallel Computing, p. 2029. Springer (2011)
43. Rul, S., Vandierendonck, H., Bosschere, K.D.: Extracting coarse-grain parallelism in general-purpose programs. In: PPOPP, ACM, pp. 281–282 (2008)
44. Stefanovic, D., Martonosi, M.: Limits and graph structure of available instruction-level parallelism. In: Bode, A., Ludwig, T., Karl, W. (eds.) Euro-Par. Lecture notes in computer science, vol. 1900. Springer, Berlin (2000)
45. Tournavitis, G., Franke, B.: Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In: PACT, ACM, pp. 377–388 (2010)
46. Wang, Z., Tournavitis, G., Franke, B., O'Boyle, M.F.P.: Integrating profile-driven parallelism detection and machine-learning-based mapping. TACO **11**(1), 2:1–2:26 (2014)

## Affiliations

**Christopher Brown[1] · Vladimir Janjic[2] · Adam D. Barwell[1] · J. Daniel Garcia[3] · Kenneth MacKenzie[4]**

Vladimir Janjic
vjanjic001@dundee.ac.uk

Adam D. Barwell
adb23@st-andrews.ac.uk

J. Daniel Garcia
josedaniel.garcia@uc3m.es

Kenneth MacKenzie
kenneth.mackenzie@iohk.io

1 School of Computer Science, University of St Andrews, St Andrews, UK

2 School of Science and Engineering, University of Dundee, Dundee, UK

3 University Carlos III of Madrid, Leganes, Spain

4 IOHK, Hong Kong, China